Picos, A Hardware Task-Dependence Manager for Task-based Dataflow Programming Models

Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, Mateo Valero
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain
Email: {xubin.tan, jbosch, mvidal, eduard, mateo.valero}@bsc.es, {calvarez, djimenez}@ac.upc.edu

Dissertation Advisor(s): *Carlos Álvarez, Daniel Jiménez-González*

DOCTORAL DISSERTATION COLLOQUIUM

EXTENDED ABSTRACT

***Abstract*—Task-based programming Task-based programming models such as OpenMP, Intel TBB and OmpSs are widely used to extract high level of parallelism of applications executed on multi-core and manycore platforms. These programming models allow applications to be expressed as a set of tasks with dependences to drive their execution at runtime. While managing these dependences for task with coarse granularity proves to be highly beneficial, it introduces noticeable overheads when targeting fine-grained tasks, diminishing the potential speedups or even introducing performance losses. To overcome this drawback, we propose a hardware/software co-design Picos that manages inter-task dependences efficiently. In this paper we describe the main ideas of our proposal and a prototype implementation. This prototype is integrated with a parallel task-based programming model and evaluated with real executions in Linux embedded system with two ARM Cortex-A9 and a FPGA. When compared with a software runtime, our solution results in more than 1.8x speedup and 40% of energy savings with only 2 threads.**

Keywords-Fine-Grain Parallelism and Architectures; Data Flow Machines; Reconfigurable Computing & FPGA Based Architectures;

I. INTRODUCTION

After reaching the limit of Dennard Scaling, parallel computing has become an ubiquitous principle to gain performance. Meanwhile, it exposes significant challenges, such as detecting parallel regions, distributing tasks/works evenly and synchronizing them. Task-based dataflow programming models are quickly evolving to target these challenges. Significant examples are OpenMP, Intel TBB, StarSs and OmpSs [1]. Using these programming models, an application can be expressed as a collection of tasks with dependences, which are managed at runtime. With moderate task sizes, those programming models are able to exploit high levels of task parallelism. However, with fine-grained tasks they suffer different degrees of performance degradation due to runtime overheads [2].

To overcome this deficiency and enable a finer task parallelism, We propose to improve the runtime by offloading some of the most time consuming runtime functions [3] (dependence analysis and task scheduling) to hardware. Our work on hardware task dependence graph management has showed great scalability and performance improvement over its software-only alternatives [2], [4], [5].

II. MOTIVATION AND RESEARCH

A. Background

OpenMP provides a powerful way of annotating sequential programs with directives to exploit heterogeneity and task parallelism. For example in C/C++ language: #pragma omp task depend(in: ...) depend(out: …) depend(inout: ...) is used to specify a task with the direction of its data dependences (scalars or arrays). Implicit synchronization between tasks is automatically managed by dependence analysis, and explicit synchronization is managed by using #pragma omp task wait, which makes a thread wait until all its child tasks finish before it can resume the code execution. We show an example of Cholesky source code with OpenMP annotations in Figure 1. When the compiler finds a task annotation in the program it outlines the next statement and introduces a runtime system call to create a task (represent as a task descriptor). At the execution time, the runtime system manages task creation, computes task dependence graph as in Figure 1, and schedules tasks when they can be executed because all their dependences are ready.

```
for (j=0; j<NB; j++) {
  for (k=0; k<j; k++)
    for (i=j+1; i<NB; i++) {
#pragma omp task depend(in: A[i][k], A[j][k])
                 depend(inout: A[i][j])
      sgemm(A[i][k],A[j][k],A[i][j]);}

  for (i=0; i<j; i++) {
#pragma omp task depend(in:A[j][i]) \\
           depend(inout: A[j][j])
      syrk(A[j][i],A[j][j]);}

#pragma omp task depend(inout: A[j][j])
  spotrf(A[j][j]);

  for (i=j+1; i<NB; i++) {
#pragma omp task depend(in: A[j][j]) depend(inout: A[i]
[j])
    strsm(A[j][j],A[i][j]);}}
```

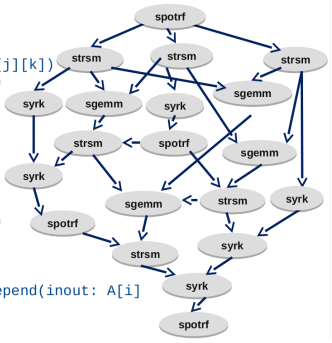

Figure 1: Cholesky Factorization

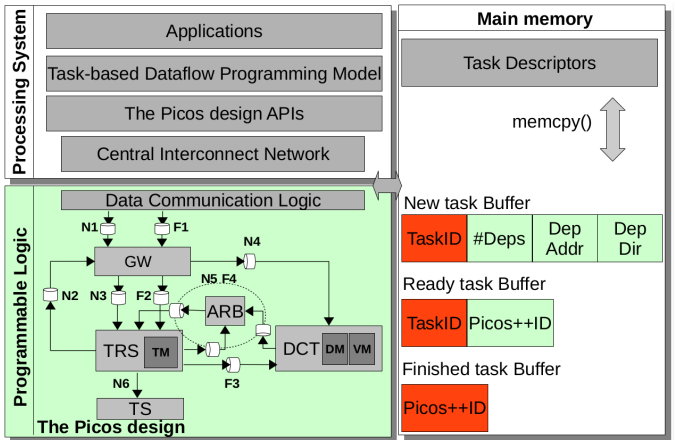

Figure 2: The Picos design in ZYNQ SoC platform

B. *Main Idea and Implementation*

The Picos design [4] manages dependence analysis and task scheduling in hardware. (1) It reads new tasks with dependences and inserts them as a node in the task dependence graph in hardware; (2) It determines if a task is ready-to-execute and schedules it to the threads; (3) It reads finished execution tasks and updates the task dependence graph.

Figure 2 shows the main organization of the implementation Picos++ on a commodity SoC platform. It mainly consists of the Processing System (with 2 ARM cores), the Programmable Logic (resides the Picos designs), the main memory. Three circular FIFOs are allocated as New/Ready/Finish task buffers to allow the communication between the 2 ARM cores and Picos++.

Picos++ [5] is composed of five main components. **Gateway (GW)** fetches new and finished tasks from outside, and dispatches them to TRS and DCT. **Task Reservation Station (TRS)** is the major task management unit. It stores up to 256 in-flight tasks, tracks the readiness of new tasks and manages the deletion of finished tasks. **Dependence Chain Tracker (DCT)** is the major dependence management unit. It stores the memory addresses of dependences as tags and performs address match for each new dependence to these arrived earlier, to track data dependences between tasks. **Arbiter (ARB)** works as a switch and multiplexes the queues between TRS and DCT. **Task Scheduler (TS)** stores all ready tasks and schedules them to idle workers. Each pair of components are connected with one or more FIFOs to ensure asynchronous communications.

Picos++ also introduced a new feature to support nested tasks in hardware. Since nested tasks are usually supported to improve the programmability and potential parallelism of applications, we consider it as a necessary feature of any task manager that executes general-purpose codes. For instance, a nested task can be found in recursive codes where the recursive function is a task, which can be further decomposed (like in H264dec and Multisort), or when using tasks to call libraries that have already been programmed with tasks. However, due to the fact that the implementations of hardware task managers, opposed to the software ones, have a limited amount of memory. It is possible that hardware task dependence managers read a task and get their internal memory full in the middle of processing task dependences which results in a system stall. Without nested tasks, this is not a problem because previous processed tasks always come first in the order of execution and eventually they finish. When these previous tasks finish their execution, they free resources that allow the hardware to proceed with the processing of the stalled task. With nested tasks, the situation is much more complicated which can lead to a system deadlock. To tackle this, Picos++ employs a hardware/software co-design mechanism which ensures atomic task processing in hardware and a buffered task recovery in software.

III. CURRENT PROGRESS

A. *Experimental Setup and Benchmarks*

Picos++ is synthesized using XILINX Vivado Design Suite 14.4 and tested in a SoC Platform which includes 2 ARM Cortex-A9 (at 666MHz) and a Programmable Logic/FPGA part (at 100MHz). We use OmpSs [1] (a OpenMP like Programming model) to integrate with Picos++. It is supported by the source-to-source Mercurium compiler 1.99 and the Nanos++ runtime system. We use Linaro Linux 14.04. Benchmarks: Cholesky Factorization (in Figure 1), Multisort and H264dec. Multisort is a variant of the mergesort algorithm. H264dec is a high performance H.264 video decoder, a video pedestrian area.h264 is selected as input.

B. *Hardware Resource and Power Consumption*

Table I shows the on-chip resource utilization and dynamic power consumption of Picos++ on the XILINX XC7Z020-CLG484 chip [6]. Picos++ and its data communication logic consume a small fraction of power around 0.02W, less than 1.3%. In addition, it uses < 20% of resource consumption, makes it feasible to be integrated in multi-core CPUs.

Table 1: Hardware resource and power consumption

Name	**Resource**			**Power**
Programmable logic XC7Z020 - Total	**LUTs** 53,200	**FFs** 106,400	**BRAM(36Kb)** 140	**Watts**
Picos++	11.32%	4.22%	18.57%	0.011
Data Communication Logic	3.90%	2.93%	0%	0.009
Processing System				**Watts**
2 ARM Cortex-A9 CPUs	-	-	-	1.53

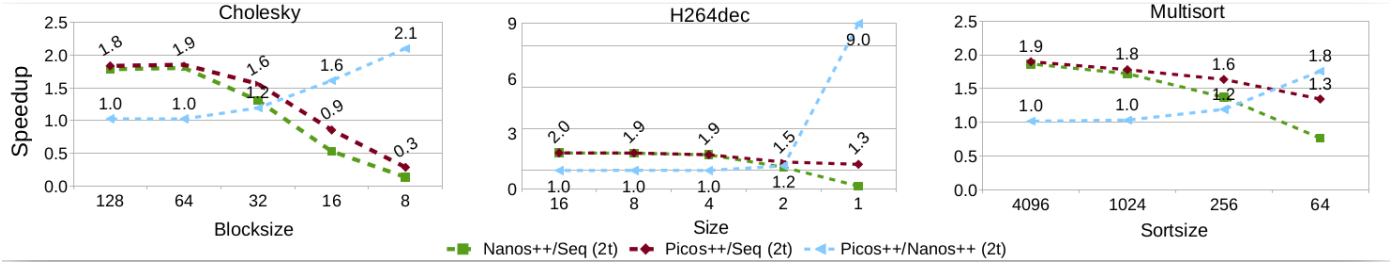

Figure 3: Speedup of real benchmarks with 2 threads

Table 2: Energy savings, #Tasks and Average task size

Name	Energy Savings	#Tasks	#AvgTaskSize(ns)
Cholesky	1% - 52%	816 - 2829056	6074213 - 4627
H264dec	1% - 89%	549 - 82310	3138500 - 32293
Multisort	2% - 42%	605 - 2397	1994309 - 219350

C. *Performance and Energy Consumption*

In this section, we present performance and energy consumption studies of Nanos++, Picos++ and Picos++ against Nanos++ with 2 threads. Figure 3 shows the speedup using OmpSs applications Cholesky (fixed problem size: 2kx2k), H264dec (fixed problem size: 10 frames) and Multisort (fixed problem size: 128K) with an decreasing task size. With smaller task size, more tasks are needed to solve the fixed size problem and thus bigger overhead to manage these tasks. Table 2 shows the energy savings for using Picos++ over Nanos++, the range of the number of tasks and the average task sizes in nanoseconds for the task size intervals showed in Figure 3.

For Cholesky, with block size 32, Picos++ achieves 1.6x speedup over the sequential version, and 1.2x over Nanos++. Correspondingly, it saves 15% of energy. With smaller block sizes, both Picos++ and Nanos++ degrade, but Picos++ degrades much slower with 2.1x speedup against Nanos++. The obtained performance results are relevant because those are for only 2 threads where it is easy to exploit all the available parallelism. For bigger systems, Picos++ will help to better exploit the strong scalability of the applications when the task granularity becomes too fine. **For H264dec**, when compared to the sequential execution, Picos++ has a 2x speedup with size 16. Moreover, with size 2, it is 20% faster than the software-only runtime Nanos++. With size 1, it saves 89% of energy. **For Multisort**, Picos++ achieves from 1.3x to 1.9x speedup over the sequential execution, and up to 1.8x over Nanos++. Correspondingly, it saves up to 42% of energy.

IV. Scalability and Future Work

The current implementation Picos++ shows good results in a fully integrated system with two available cores. Previous design exploration with 24 threads and no software integration, proved to have great scalability for up to 21x speedup for Cholesky [2]. Moreover, by using a software cycle-level simulator [4] it has been measured that the same design with a larger size is able to manage up to 256 workers.

During the executions, we also observed that the threads are wasting a good amount of time busy checking for ready tasks when there is no parallelism in the application. We measured one execution with multisort and concluded that it is possible to save an additional 10% of energy by exploring Picos++ to hint the threads into low power mode [5].

Another interesting future work would be to use Picos++ to manage task scheduling and data movements for general-purpose processors and other heterogeneous cores. This allows to study the scalability of Picos++ with more execution units and also save the usage of threads dedicated to manage the data movements to special hardware execution units like GPU.

V. Conclusion

In this paper we show a brief description of the Picos design, as a high speed, small and energy efficient runtime accelerator for task-based dataflow programming models like Open MP 4.0. The presented implementation Picos++ has been evaluated with real applications executing in a Linux embedded system with two ARM Cortex-A9 cores and a FPGA. Results show that the prototype greatly outperforms the existing software-only runtime and save up to 89% of energy consumption with only 2 threads. More importantly, with a larger design with multiple task and dependence management units upcoming, it would able to exploit a larger magnitude of parallelism in the applications with very fine granularity, that software alternatives cannot achieve.

Acknowledgment

This work is supported by the projects SEV-2015-0493 and TIN2015-65316-P, by the project 2014-SGR-1051 and 2014-SGR-1272, by the RoMoL GA 321253 and by the project cooperation agreement with LG Electronics, and thank the Xilinx University Program.

References

[1] B. S. Center, “Ompss programming model,” [online], 2016, https://pm.bsc.es/ompss.

[2] X. Tan, J. Bosch et al., “Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models,” Int. Symp. on Performance Analysis of Systems and Software (ISPASS), 2016.

[3] N. Engelhardt, T. Dallo et al., “An integrated hardware-software approach to task graph management,” in In 16th IEEE Int. Conf. on High Performance and Communications(HPCC-2014), 2014.

[4] F. Yazdanpanah, C. Alvarez et al., “Picos: A hardware runtime architecture support for ompss,” Future Generation Computer Systems(FGCS), 2015.

[5] X. Tan, J. Bosch et al., “General purpose task-dependence management hardware for task-based dataflow programming models,” 31st International Parallel and Distributed Processing Symposium (IPDPS), 2017.

[6] XILINX, “Zynq-7000,” [online], 2015, https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html